Question 1

Other Solution

In this problem, the player starts at a designated point ('i') and needs to reach another designated point ('j') and the task is to make the minimum adjustments to the current airflows or spikes.

Inspired by the hint, my rough idea is to consider each cell as a vertex that connects to its nearby (up, left, down, right) cells. If the nearby cell is at the direction of the airflow, then set the weight of the original cell pointing to it as 0, otherwise (not in the direction or spikes) set the weight as 1; and then use Dijkstra's algorithm to find the shortest path length. This way, the length would be the minimum adjustments needed, since in a path the cells that needs to change (not in the airflow direction or spikes) has the weight of 1 and that needs not to change has a weight of 0, and we have minimized the total weight.

My original program is to first create a graph representation and then do Dijkstra's algorithm based on it. This will have a time complexity of O (mn*log(mn)). However, when running this on OJ it will exceed the memory limit in large inputs. It has space complexity of O(mn), but the constant can be large.

My Solution

To improve the memory, I implemented a second version program that did not pre-store the graph information. I used arrays to store the possible four directions and their corresponding coordinates and used java's data structure PriorityQueue to implement the Dijkstra's algorithm, also I rewrite the compareTo function to compare node's cost. And in Dijkstra function, the current adjacent vertex is get by going through all four directions and at the same time the weight is calculated. Then do relaxation for each adjacent vertex and update the priority queue. and when it reaches the end, return the distance ('currCost').

The time complexity is O(mn*log(mn)), and the space complexity is O(mn), which is the same as the previous solution. However, the constant should be much smaller and there's no exceeding memory limit problem now.

Question 2

My solution

This task involves processing a given graph with n vertices and m edges, adding new edges based on specific condition and then perform BFS from a given starting vertex. The condition is for every vertex I, if it has two neighbors u and v such that u = k*v or v = k*u, then add a new edge between u and v if there's no edge before.

In my program, I used 'add_edges' function for edge addition and 'bfs_traversal' function for BFS. The add_edges function iterates through every vertex and examines the k× relationship between every pair of neighboring vertices. If the condition is met and there is no existing edge, it adds an edge between them. The bfs_traversal implements the standard BFS algorithm with an additional sorting step to ensure that neighbors are traversed in ascending order of their IDs. It uses a deque data structure

in python to manage the order of traversal so the enqueue and dequeue have O (1) time complexity, and a visited array to keep track of already visited vertices. The result is a list of vertices in the order they were visited by BFS.

Then in solve function it takes the input, build the graph, add the edges, and print the result. This program have a time complexity of O(n^3) (add_edges function have O(n*d^2), where d is the maximum degree, which is n; bfs function's time complexity is dominated by the sorting, which is O(n*dlog(d))). This program is straight forward and passed OJ, so I didn't think of other solutions.